Dynamic Decision-making with Reflecting and Learning for Self-adaptive Systems

Misoo Kim 1, Hohyeon Jeong 1, Eunseok Lee 1

1Information and Communication Engineering, Sungkyunkwan University, Suwon, Republic of Korea

Abstract - Self-adaptive systems can change their sub-goals and behaviors to achieve an ultimate goal in a changing environment. Existing approaches can be adapted in environments using pre-defined utility functions and human strategies; however, human designers cannot perfectly assume and predict all possible system environments during design time. We propose a new method of dynamic decision-making for real-time adaptive strategies through the following steps. First, we design a dynamic decision network (DDN) with environmental factors and goal model. Second, we evaluate and predict goal satisfaction using the DDN. We further propose a dynamic reflection method that changes the model using real-time data. We applied the proposed method in Robocode and verified its effectiveness by comparing it with static decision-making.

Keywords: self-adaptive system, dynamic decision-making, dynamic decision network, reflection model

1 Introduction

Self-adaptive systems (SASs) can change their own sub-goals and behavior without a human operator to achieve an ultimate goal in a variety of environments [1]. To change sub-goals and behavior, researchers have proposed goal-model-based SASs because they provide information for the evaluation of goal satisfaction and the formulation of rational decisions [1–2].

Existing decision-making techniques for goal-model-based SASs have pre-defined utility functions and the strategies depend on assumptions at the design time [3–9]. However, during the design time, system designers cannot perfectly assume and predict all possible system environments that the system will be deployed in; as a result, neither goal achievement nor proper adaptability can be guaranteed. To cope with this limitation, SASs require dynamic-decision-making that can consider the variety of runtime environments that can be known after deployment.

Among Artificial Intelligence (AI) techniques, the dynamic decision network (DDN) [10], which is used in our research, is a proper model for the making of dynamic decisions. It can design alternatives and outcome-utility values for all the alternatives using environmental information. Consequently, it can make the best possible decisions within changing environments.

In this paper, we propose a method of dynamic decision-making for the deployment of SASs in runtime environments that are not known prior to deployment. To represent the system goals and environment, we designed a DDN using the goal model and environmental information so that decision-making can occur during runtime. After system deployment, our DDN can evaluate and predict each goal satisfaction during runtime using real-time environmental information. Furthermore, the DDN is dynamically reflective, so it can make suitable decisions continuously in a variety of runtime environments. SASs can therefore adapt appropriately to their various runtime environments through the use of the proposed method, which considers both the real system environment and goal satisfaction, even when the environmental knowledge of the system designers is imperfect. We applied our proposed method in the Robocode context [11] and verified its effectiveness through a comparison with the conventional static-decision-making method.

The rest of the paper is organized as follows: Section 2 describes related work; Section 3 presents the proposed approach; and in Section 4, we describe how we applied the proposed method in Robocode and evaluated it using experiments, thereby allowing for the presentation of our conclusions in Section 5.

2 Related Works

2.1 Decision-making in Goal-model-based SASs

The goal model represents system goals according to the stakeholders’ objectives captured by Requirement Engineering. Figure 1 depicts a goal model for the Robocode robot, which is utilized in our research. It is composed of the goals (a circle) that should be achieved for the system, tasks (a hexagon) that form the operations of the system for the achievement of the goals, and links (a solid-arrow) that indicate the goal refinement achieved by facilitating connections via AND/OR decompositions. To satisfy a decomposed goal, all/at least one of the sub-goals should be satisfied through an AND/OR link; therefore, alternatives can be captured in accordance with an OR decomposition. Goals can be divided into hard goals, functional requirements with conditions (a dotted-rectangle)
[3] that represent satisfactory goal conditions, and soft-goals, non-functional requirements.

For example, to satisfy the goal “Move for Avoidance” at the bottom-left side of Figure 1, the condition “Event=hitBy" must be satisfied, which prevents the robot from being hit by an enemy missile. Furthermore, to satisfy that condition, the robot chooses a task from among “Random Movement,” “Normal Movement,” and “No Movement” based on its decision-making. The goal model provides information for the evaluation of goal satisfaction, and rational decisions are consequently made regarding the OR decomposition [1-2]. Therefore, we use it in this study to create adaptation strategies.

The decision-making techniques of existing goal-model-based SASs evaluate or predict goal satisfaction using pre-defined utility functions, which allows them to determine adaptation strategies regarding their runtime environments [3–8]. Morandini et al. [3] introduced the concept of Condition for the representation of satisfactory goal conditions on the environmental level; because it can be mapped with environmental factors, we adopted this method. Baresi et al. [4] and Nunes et al. [5] used pre-defined utility functions to make decisions in handling the uncertainties of soft-goal satisfaction. Menasce et al. [6] used a linear utility function with a static threshold that can be defined according to goal satisfaction. Welsh et al. [7] proposed a Claim for the expression of an environment that is assumed on a goal model during the design time. The previously mentioned research studies, however, would lead to the making of unsuitable decisions in real time because none of them can predict the adaptation result after executing a strategy. Chen et al. [8] predicted goal satisfaction based on the probability of success or failure. To adapt to the runtime environment, they proposed an updating probability method regarding goal satisfaction and an evaluation method regarding overall system utility values using [12]. With the exception of the real environmental information that can affect goal satisfaction, their approach considers only an abstract goal model. In our previous research [13] on adapting to the real system environment, we used a DDN in the design for system-environment information to illustrate both a deployed environment and goals.

2.2 DDN-based Decision-making

The DDN [10] is a proper AI technique for making dynamic decisions. Figure 2 shows a DDN with its corresponding components and time slices. The DDN uses a time slice to extend decision networks (DNs), using temporal dependencies (a dotted arrow) to express changes within the time domain. The DNs are composed of decision nodes (a rectangle) for a set of alternatives, utility nodes (a diamond) that compose a utility table (UT) to estimate the outcome-utilities of the alternatives, chance nodes (a white circle) such as random variables with a conditional probability table (CPT) for the occurrence probabilities of events, and causal dependencies (a solid arrow) that exist between the nodes; here, chance nodes are inferred for the next time slice through observed evidence nodes (a gray circle). The system environmental information (EI) can be designed using the set of chance nodes about environmental factors (EFs) and their dependencies. A DDN supports decision-making as well as evaluation of goal satisfaction for SASs because it can predict EI and the outcome-utility values of decisions at varying times.

The DDN has already been verified as part of a dynamic-decision-making method for a variety of domains: the auto-defense system for an aircraft [14], a groundwater management system with a climate-prediction capability [15], and an assisted-living system that assigns caregivers to an elderly man [16]. In the corresponding research, a DDN used to predict EI makes decisions for a specific domain and not for a variety of domains; however, we propose a general purpose DDN-modeling process derived from the goal model and EI.

For SASs, Bencomo et al. [9] proposed a modeling framework that transforms the goal model into a DDN for decision-making by SASs. The framework can predict soft-goal satisfaction based on a pre-defined probability in the abstract goal model; the pre-defined probability should lead to the making of suitable decisions in real time. The application of such a DDN concept is limited to the use of runtime EFs for monitoring design-assumption deviation, which is to say that it considers only a partial goal model that contains soft goals and tasks. The research focus of the previous studies and our focus differ in how they handle design assumptions. That is, our current work considers both the runtime-EI and the overall goal model in the design of the DDN, extending our previous research [13], which was limited to the design of soft-goals.
3 Proposed Approach

3.1 Goal-model-based DDN-modeling

In this section, we describe a modeling process that uses a DDN to fill the gap between the goal model and the system by considering the deployed environments, as shown in Figure 3. This method extends our previous paper [13] and integrates its mapping for soft goals.

A DDN structure can abstractly design decision making model using a goal model. First, we narrow down the overall goal model into a compacted goal model consisting of leaf goals that comprise only an OR decomposition. Second, we map the compacted goal model onto the DDN. The top of Figure 3 explains how the goal model can be abstractly mapped onto the DDN. The tasks, conditions, soft goals, and hard goals in the goal model are mapped with the decision, evidence-based chance, static chance, and utility nodes in the DDN, respectively. Third, to describe the EI, we extend the abstract DDN to add the EFs that have causal relationships with EFs expressed within conditions and to add temporal dependencies with pre-defined time slices between the same EFs. Thereby, a DDN has the EI, which is composed of EFs and the causality among them, which is helpful in representing the real system environment. Fourth, to build a CPT of a DDN, the system designers define the discretization criteria to represent the states of the chance nodes and subsequently build the CPT with those discrete values using the knowledge of domain experts or learning algorithms. In the design of a complex system, however, the number of EFs would be too great, making the inference time too long. To deal with that problem, designers can use a feature-selection algorithm, such as that in [17], to select important EFs. We suggest a mutation information–based feature select ion algorithm such as that in equation 1, where CEF is an EF for a specific condition, and EF is all EFs except CEFs. A pre-defined threshold can be used to decide which EFs to use as CEFs.

\[ I(CEF;EF) = \sum p(CEF,EF) \log \left( \frac{p(CEF,EF)}{p(CEF)p(EF)} \right) \]  

Fifth, to build a UT of for a DDN, according to the experts or the historical data, we design the UTs of the leaf-utility nodes are designed according to the outcome-utility values, depending on the goal conditions and each alternative. The parent-utility nodes comprise UTs with sub-goal weights for the satisfaction of their own goals.

3.2 Dynamic Decision-making Techniques

In terms of dynamic decision-making, our approach maintains and updates a DDN that is stored in the Knowledge base. Figure 4 presents an overview of the proposed framework. Algorithm 1 programmatically describes the proposed dynamic decision-making techniques in our framework.

Algorithm 1 Dynamic Decision Making with Reflection Model

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDN, MonitrobleData, HistoricalData, a, b</td>
<td></td>
</tr>
</tbody>
</table>

1. // 1. Reflecting Layer
2. currentState = discretize(MonitorableData)
3. while i < DDN.changeNode.length() do
4. stateNumber = DDN.changeNode.get(i).stateNumber()
5. while j < stateNumber do
6. state = chanceNode.get(j).get(Sate())
7. if currentState.contains(state) then
8. state = state.getProbability() + 1/10
9. else
10. state = state.getProbability() - \frac{1}{10} > selfNumber - i
11. end if
12. end while
13. DDN.applying(state)
14. i = i + 1
15. end while
16. /2. Learning Layer
17. currentUtility = evaluator(MonitorableData)
18. if currentUtility != predictedData.getUtility() then
19. previousDecision = previousData.getPreviousData()
20. if previousDecision != DDN.attitudeNode.getTable()
21. while i < previousUtility.getNumberOfList() do
22. table = DDN.attitudeNode.getTable()
23. historicalUtility = historicalUtility + matrixValue * currentUtility
24. DDN.updatingTable()
25. i = i + 1
26. end while
27. end if
28. /3. Adaptation Layer
29. utilityList = calculateUtility(DDN)
30. ordering(utilityList)
31. return utilityList.getTheHighestUtility()
The Monitor regularly collects the raw data that represents the EI of the target system. According to the need, it also transforms the raw data into discrete values using discretization criteria.

The Analyzer checks whether the predicted EI matches the real EI. If a difference is found, it calculates and updates the new conditional probability in the CPTs using equations 2 and 3 for the CPTs of child chance nodes, and equations 4 and 5 for the CPTs of root chance nodes based on the Dirichlet Distribution [18] and our previous research [13]. We use the CPT to predict the EFs in the runtime environment and represent the EI in the Planner in the Adaptation Layer.

\[
P(EF_i = x | EF_{j}, \ldots, EF_n) = \text{previous} P(EF_i = x | EF_{j}, \ldots, EF_n) + \frac{1}{10^n} \tag{2}
\]

\[
P(EF_i = x | EF_{j}, \ldots, EF_n) = \text{previous} P(EF_i = x | EF_{j}, \ldots, EF_n) + \frac{1}{10^n} \times 1/(m-1) \tag{3}
\]

\[
P(EF_i = x | EF_{j}, \ldots, EF_n) = \text{previous} P(EF_i = x) + 1/10^n \tag{4}
\]

\[
P(EF_i = x | EF_{j}, \ldots, EF_n) = \text{previous} P(EF_i = x) + 1/10^n \times 1/(m-1) \tag{5}
\]

According to α, which indicates how much of the current state needs to be reflected, equations 2 and 3 show a way to update the conditional probability of the \(i\)-th EF (EF) using \(m\) discrete values given from the \(j\)-th (ENV\(_j\)) to the \(n\)-th (ENV\(_n\)), which are the parent nodes of the \(i\)-th EF. Using the previous CPT, equation 2 is used to increase the probability of the currently evaluated discrete value \(x\), and equation 3 is used to decrease the probability of the remaining \(m-1\) discrete values, with the exception of the current discrete value. Furthermore, the CPT regarding the temporal dependency can be updated using both equations 2 and 3. Given that the current time slice is \(i\), information for time slices from \(j = i-1\) to \(n = i-t\) and \(t < i\) were available for this study. The root node indexed by \(k\), which does not have parent nodes, would be updated using equations 4 and 5; accordingly, CPTs would be maintained on a variety of runtime environments. In addition, previous CPTs are shifted to the \(t-1\) time slice. After reflecting the real environmental information in CPTs, the Applier applies the new CPTs to the DDN of the target systems.

### 3.2.2 Learning Layer

The Learning Layer comprises the Observer, Evaluator, and Updater to learn the utility values for the executed decisions and real environmental information.

The Observer step regularly observes the completion of executed decisions and collects the raw data. The Evaluator calculates the real utility values and checks whether the predicted utility values and real utility values are the same after executing decisions.

The Updater calculates and updates the new utility values for the UTs using equations 6 and 7, based on our previous research [13]. In our approach, the UT is used to predict the utility value through each alternative and to calculate the expected utility in the Planner in the Adaptation Layer. It is difficult to know to what degree and by which action the system acquires the overall utility value (OV); to estimate this, we used the weights of the parent-utility nodes, which explain the priorities of the lower goals to the upper goals. Equation 6 illustrates how a residual value is divided between the predicted utility value at a previous time (predicted(OV)) and the real-time utility value (real(OV)) into a partial utility value according to the weight of the \(r\)-th parent-utility node with respect to the \(q\)-th leaf-utility node (\(U(U_q)\)). Equation 7 represents the predictive utility value of the \(i\)-th utility node through a previous decision (D).

\[
V(U_q) = (\text{real(OV)} - \text{predicted(OV)}) \times U(U_q) \tag{6}
\]

\[
U_d(D | EI) = \text{predicted} U_d(D | EI) + \text{real} (U_d) + \sum_{i=0}^{k} V_i (U_d)/(k+1) \tag{7}
\]

### 3.2.3 Adaptation Layer

The Adaptation Layer comprises the Detector, Planner, and Effector to make adaptive decisions using reflected and learned DDN.

The Detector step periodically senses the set of raw data and calculates the real utility values.

The Planner infers the subsequent EFs and obtains probabilities regarding the discrete values of the goal-satisfaction conditions. At the current time \(t\), equation 8 is used to predict the EFs (X) for the next time (X\(^{t+1}\)), which composes a parent EF (Y) and the current \(i\)-th EF (X\(_i\)) when each alternative (D\(_{i-1}\)) is performed, based on the Markov property [10]. The Planner then calculates the expected utility values for each strategy according to equation 9, followed by the formulation of a decision that includes the highest expected utility, which is sent to the Effector.

\[
P(X|Y, D_{i-1}) = P(X|Y, Y^{t+1}, D_0)P(X|Y^{t+1}, Y, D_{1:i}) \tag{8}
\]

\[
\text{ExpectedUtility} = \sum_{d=1}^{m} U_d(U)|U_d) \times \\
\sum_{i=1}^{n} U_i (X^{t+1}|D_{i-1}) \times P(X^{t+1}|X, Y^{t+1}, D_{i-1}) \tag{9}
\]
4 Case Study

4.1 ROBOCODE

To verify the effectiveness of the proposed method, we applied it in the context of Robocode [11], a Java-based programming game wherein virtual robots battle against teams or individuals. When a battle is started, it can be watched, along with the result and score after its end, on a screen. The game is thus advantageous for quantifying adaptation results.

A robot consists of a Vehicle Heading for movement; Missile Heading for a turret that fires a missile to decrease the energy of oneself or an enemy; and a Radar Heading to detect enemies and collect their information. These components are controlled independently, so they form the basis of a strategy.

During a battle, a robot can obtain two types of information, as shown in Table 1. If the radar sensors an enemy robot, it collects enemy information. The robot can use that information to make decision. After firing a missile, which disappears from the screen after it hits or misses an enemy, the robot receives a Hit Event or Miss Event; alternatively, if a robot is hit by a missile, it receives a HitBy Event.

A robot might not use an appropriate strategy to avoid an enemy missile or accurately attack an enemy because it cannot collect perfectly accurate information about the time that an enemy fires a missile and the missile’s trajectory. The robot therefore needs to be able to predict the enemy state in a variety of systemic runtime environments given unknown robots that can be known only after a battle.

Table 1: Obtainable Information

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
<th>Data Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enemy Information</td>
<td>Distance</td>
<td>Distance from my robot to scanned robot</td>
<td>0 to diagonal length of field</td>
</tr>
<tr>
<td></td>
<td>Heading</td>
<td>Vehicle heading</td>
<td>0° to 360°</td>
</tr>
<tr>
<td></td>
<td>Bearing</td>
<td>Direction relative to my current heading</td>
<td>-180° to 180°</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>Remaining energy</td>
<td>0 to 100, 100 to 100 to 180°</td>
</tr>
<tr>
<td></td>
<td>Velocity</td>
<td>Velocity (negative is backward)</td>
<td>-8 to 8 (pixel/sec)</td>
</tr>
<tr>
<td>Event</td>
<td>Hit</td>
<td>My bullet hit the target</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HitBy</td>
<td>Hit by an enemy missile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Miss</td>
<td>My fired bullet missed</td>
<td></td>
</tr>
</tbody>
</table>

4.2 Experimental Setup

4.2.1 Goal-model-based DDN Modeling

Each goal condition is designed according to Table 2.

Table 2: Goal Satisfaction Criteria

<table>
<thead>
<tr>
<th>Goal</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoidance</td>
<td>Event != HitBy</td>
</tr>
<tr>
<td>Attacking</td>
<td>Event = Hit</td>
</tr>
<tr>
<td>Targeting</td>
<td>enemyVectorX = realVectorX</td>
</tr>
<tr>
<td>(control)</td>
<td>enemyVectorY = realVectorY</td>
</tr>
</tbody>
</table>

We chose the robot “EpeestMicro,” which was recently published by the community [20], and used it as a base-bot. We modeled the DDN using OpenMarkov [21] to express a design assumption, which we collected until we had 10 rounds of base-bot information. The battle type is a man-to-man fight. We built the EI of a DDN using the hill climbing algorithm and Bayesian scoring, with the historical base-bot information serving as the basis. We discretized each EF in reference to [22]. Each factor affects itself on the next occasion, when it is used as an evidence node.

According to the mapping method shown in Figure 3, we designed each UT of for leaf-utility nodes using the historical data scores. We designed the decision nodes in the DDN using the tasks in Table 3, which shows the robot’s goals and tasks from Figure 1 and whether the base-bot considers the goals and tasks. We designed the tasks of our robot from the base-bot, but the execution points of our tasks are different from those of the base-bot because we used the dynamic EI—that is, the dynamic-utility function—to evaluate goal-satisfaction.

Table 3: Basic Strategies for Goals

<table>
<thead>
<tr>
<th>Goal</th>
<th>Tasks</th>
<th>Base-bot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move to</td>
<td>Random</td>
<td>O</td>
</tr>
<tr>
<td>Avoid Miss</td>
<td>Normal</td>
<td>O</td>
</tr>
<tr>
<td>Accurate</td>
<td>Current</td>
<td>X</td>
</tr>
<tr>
<td>Targeting</td>
<td>K-NN</td>
<td>O</td>
</tr>
<tr>
<td>(Control)</td>
<td>DBN</td>
<td>X</td>
</tr>
<tr>
<td>Effective</td>
<td>Min. power</td>
<td>X</td>
</tr>
<tr>
<td>Attacking</td>
<td>Average power</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td>Max. power</td>
<td>X</td>
</tr>
</tbody>
</table>

The number of time slice $t$ was set to 3, and we used three pieces of historical evidence to infer the factors on the next occasion.

4.2.2 Dynamic Decision-making

We designed the parameters shown in Table 4.

Table 4: Parameter Settings

<table>
<thead>
<tr>
<th>Layer</th>
<th>Module</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reflecting</td>
<td>Analyzer</td>
<td>$\alpha = 4$</td>
</tr>
<tr>
<td>Learning</td>
<td>Evaluator</td>
<td>Overall utility = real-time score</td>
</tr>
<tr>
<td>Updater</td>
<td>$k = 4$</td>
<td></td>
</tr>
</tbody>
</table>

4.3 Evaluation Method

Our method can predict goal satisfaction, whereby we use events to determine an avoidance and firing strategy and “enemyVectorX/Y” to control the missile and radar heading in Table 2, according to whether each condition will be satisfied. The prediction accuracy is 100 % when a prediction result is correct; otherwise, the prediction accuracy is 0 %. We used the following three steps to evaluate the proposed method.

First, to evaluate the effectiveness of the DDN-based prediction, we compared the average prediction accuracies of the base-bot and the DDN-based bot, excluding the reflecting and learning layers. Second, to identify whether the reflecting layer provides adaptability in a variety of runtime environments that weren’t considered during design, we compared the...
average prediction accuracies of the DDN-bot and the R-DDN-bot based on the reflecting layer using a battle with sample robots not considered during the design. Last, to verify the effectiveness of our overall framework for dynamic decision-making, we compared the score results of the base-bot with those of the Adaptive-DDN-bot based on our framework (A-DDN-bot).

4.4 Evaluation Results

4.4.1 Effectiveness of DDN-based prediction

This section illustrates that the DDN-bot designed using base-bot information can accurately predict goal satisfaction. Table 5 shows a comparison of the average prediction accuracies with the base-bot: the prediction accuracy of the DDN-bot is higher than that of the base-bot.

Table 5 Accuracy Comparison of Goal Satisfaction Predictions

<table>
<thead>
<tr>
<th></th>
<th>Base-bot</th>
<th>DDN-bot</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>53.9 %</td>
<td>72.2 %</td>
<td>+ 34.0 %</td>
</tr>
<tr>
<td>Control</td>
<td>70.2 %</td>
<td>74.0 %</td>
<td>+ 3.8 %</td>
</tr>
</tbody>
</table>

When it scans an enemy, the base-bot predicts that enemy’s next position using a linear prediction based on a predefined utility function. Alternatively, the DDN-bot uses an EI-based prediction. The base-bot assumes that enemies move linearly, but the DDN-bot assumes that enemies select a movement strategy according to their own current information.

For example, the base-bot fires a missile of average power to the predicted position of the enemy whenever it scans an enemy, meaning that the base-bot uses a pre-defined strategy. The DDN-bot predicts events such as the “Hit” or “Miss” of a missile; furthermore, it fires a missile with low or high power when the probability of a “Miss” or “Hit” is high, respectively. In addition, the base-bot moves to avoid a missile fired by an enemy when the enemy’s energy decreases, but an enemy’s energy can decrease for any reason, including ramming into a wall. The DDN-bot predicts an event, such as “HitBy,” with respect to an enemy’s missile based on the current enemy’s information. As a result, the DDN-bot considers much more information, resulting in higher accuracies than the base-bot regarding the goal “control” and events. The effect on “control” is lower than that on “event” because the base-bot uses k-NN prediction that is effective on continuous movement.

4.4.2 Effectiveness of Reflecting Layer

First, we compared the average prediction accuracies of the DDN-bot and the R-DDN-bot, as shown in Table 6.

Table 6 Accuracy Comparison Between DDN and R-DDN

<table>
<thead>
<tr>
<th></th>
<th>DDN</th>
<th>R-DDN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>72.2 %</td>
<td>76.8 %</td>
</tr>
<tr>
<td>Control</td>
<td>74.0 %</td>
<td>77.5 %</td>
</tr>
<tr>
<td>Effect</td>
<td>+6.7%</td>
<td>+4.7%</td>
</tr>
</tbody>
</table>

Next, to identify the adaptability according to a variety of runtime environments, the DDN-bot and R-DDN-bot fought against other bots that we didn’t consider during the design time. Figure 5 illustrates the trend regarding accuracy during the battles with bots not considered in the DDN. Table 7 shows the results of a battle between the R-DDN and the static DDN, revealing that the R-DDN-bot is much more effective than the DDN-bot across a variety of runtime environments.

Table 7 Accuracy Comparison in Battle Environment with Other Robots

<table>
<thead>
<tr>
<th></th>
<th>DDN</th>
<th>R-DDN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>59.6 %</td>
<td>62.5 %</td>
</tr>
<tr>
<td>Control</td>
<td>63.4 %</td>
<td>62.5 %</td>
</tr>
</tbody>
</table>

The results show that although the DDN was modeled based on a specific robot (base-bot), it successfully used real-time information.

4.4.3 Effectiveness of Dynamic Decision-making

Having verified the reflecting layer, we show the higher scores of the A-DDN. Table 8 explains the battle result for fights between the base-bot and the A-DDN-bot. After 10 rounds in 10 battles, both the total score and the bonus score of the A-DDN-bot are higher than those for the base-bot.

Table 8 Battle Result with Base-bot

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Survival</th>
<th>Survival Bonus</th>
<th>Damage</th>
<th>Damage Bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base-Bot</td>
<td>3451</td>
<td>1200</td>
<td>245</td>
<td>1801</td>
<td>205</td>
</tr>
<tr>
<td>A-DDN</td>
<td>3937</td>
<td>1455</td>
<td>298</td>
<td>1939</td>
<td>245</td>
</tr>
<tr>
<td>Effect</td>
<td>+486</td>
<td>+255</td>
<td>+53</td>
<td>+138</td>
<td>+40</td>
</tr>
</tbody>
</table>

Table 9 compares the results from the battles with the other bots with those from the battle with the A-DDN-bot. For the proposed method with dynamic decision-making, the enemy robot score decreased by approximately 6.7%, while our robot’s score increased by approximately 13.0%. From Table 8 and Table 9, it is evident that the results for the robot with dynamic decision-making and the proposed method are more favorable than those for the static robot.
5 Conclusion and Future Work

A limitation of the existing decision-making protocol for SAs has been that neither goal achievement nor proper adaptability could be guaranteed because the design assumptions inadequately consider the execution environment. To solve this problem, we have here proposed a dynamic-decision-making technique in the design method using a goal-model-based DDN and a new framework.

We validated the effectiveness of our proposed method using Robocode. According to our experimental results, the prediction accuracy of the base-bot with a static utility is lower than that of the environmental-information-based DDN-bot. Using the proposed method, the accuracy of the model with an inadequate design assumption increases, and in other deployment environments, the changing accuracy trend increases. Also, in comparison with the static DDN, the prediction accuracy for goal satisfaction increased by an approximate average of 13.45 \%. Last, the A-DDN-bot obtained a higher score than the static-utility-based bot, and the enemy robot score decreased by approximately 6.7\%, while our robot’s score increased by 13.0\%.

In the proposed method, the environmental factors are determined according to all the information that can be obtained in the environment; however, the monitoring of the overhead must also be shown. Our future work will therefore study preprocessing and a design process to decrease the overhead. Moreover, our approach should be validated in various case studies.

6 Acknowledgements

This research was supported by the Next Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) and is funded by the Ministry of Education, Science and Technology (No. 2015045358) and the MISP (Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in Software, which is supervised by the IITP (Institute for Information & Communications Technology Promotion).

7 References


<table>
<thead>
<tr>
<th>Enemy Bot</th>
<th>vs. Base-Bot</th>
<th>vs. A-DDN-Bot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enemy</td>
<td>Mine</td>
<td>Enemy</td>
</tr>
<tr>
<td>SpinBot</td>
<td>32</td>
<td>1387</td>
</tr>
<tr>
<td>Corner</td>
<td>36</td>
<td>1345</td>
</tr>
<tr>
<td>FireBot</td>
<td>23</td>
<td>1471</td>
</tr>
<tr>
<td>Crazy</td>
<td>104</td>
<td>1614</td>
</tr>
<tr>
<td>Average</td>
<td>48.75</td>
<td>1454.25</td>
</tr>
<tr>
<td>Effect (%)</td>
<td>-6.7%</td>
<td>+13.0%</td>
</tr>
</tbody>
</table>

Table 9: Score Result Comparison with Other Bots